RAPPORT DE STAGE :

Adaptation au multicast d'un logiciel d'appartenance de groupe et de diffusion à ordre total tolérant aux fautes

Alexandra COPPÉ

mai-juin 2006

Table des matières

1 Introduction **3**

2 Protocole de groupe et de diffusion **4**
2.1 Présentation générale 4
2.1.1 Protocole de groupe 4
2.1.2 Diffusion à ordre total 5
2.1.3 Résistance aux fautes 5
2.2 Présentation de l'algorithme 5

3 Présentation du programme **7**
3.1 Explication du fonctionnement et du code du programme . . 7
3.1.1 Intégration au groupe 7
3.1.2 L'utilisateur 8
3.1.3 Le thread de réception 9
3.1.4 Le thread de phase 9
3.2 Les threads 11
3.3 Adresses IP et sockets 12
3.4 Participation personnelle 12

4 Protocole réseau : Multicast **13**
4.1 Présentation du multicast et intérêt dans notre cas 13
4.1.1 Présentation 13
4.1.2 Intérêt 14
4.2 Code test 14
4.2.1 Code initial 14
4.2.2 Tests 15
4.2.3 Modifications et conséquences 15
4.3 Modification du code 16
4.3.1 group_admin 16
4.3.2 Intégration au groupe 17
4.3.3 Envoi de messages 21
4.3.4 Divers 22
4.4 Fonctionnement et tests 23

4.4.1 Fonctionnement 23
4.4.2 Tests 24

5 Conclusion **25**

A Programme test multicast **26**

B Classe group_admin **31**
B.1 Le .h 31
B.2 Le .cpp 31

C total_order_channel **33**

D Initiateur **34**
D.1 Constructeur multicast_channel approprié 34
D.2 read_socket 35

E Suiveur **38**
E.1 ouv_sock_INIT 38
E.2 init_membership 39
E.3 recept_INIT 40
E.4 receive_INIT 40

F Initialiseur **42**
F.1 start_join 42

G Envois de messages **44**
G.1 broadcast 44

H Fonctionnement du programme **45**
H.1 En tant qu'initiateur 45
H.2 En tant que suiveur 46
[1]

[1] le code donné en annexe n'est pas exhaustif, seules les parties les plus intéressantes des méthodes présentant réellement un intérêt sont présentes

Chapitre 1

Introduction

J'ai fait mon stage au sein du LAAS (Laboratoire d'Analyse et d'Architecture des systèmes) du CNRS à Toulouse, dans le groupe de recherche tolérance aux fautes et sureté de fonctionnement informatique. Employant entre 500 et 530 personnes, le LAAS est le plus gros laboratoire du CNRS, en raison de ses nombreux partenariats, dans l'industrie notamment, il est également tourné vers les applications. Ses activités sont très diversifiées, il comprend en effet quatre principaux pôles de recherche :

- le pôle Micro et nano systèmes (MINAS)
- le pôle Modélisation, Optimisation et Conduite de Systèmes (MOCOSY)
- le pôle Robots et Systèmes Autonomes (ROSA)
- le pôle Systèmes et informatiques Critiques (SINC)

Mon stage a consisté à adapter au protocole multicast l'interface réseau d'un logiciel d'appartenance de groupe et de diffusion à ordre total tolérant aux fautes en cours d'élaboration. Pour cela j'ai dû me familiariser avec le programme déjà existant (comprendre son fonctionnement, sa manipulation), les différentes notions informatiques utilisées (ne serait ce que le langage de programmation) et enfin acquérir des connaissances suffisantes en "multicast" afin de pouvoir l'implémenter. Il s'est ensuite agi d'étudier et de prévoir l'ensemble des modifications nécessaires, puis d'implémenter ces modifications dans le code. Et enfin il a fallu tester le programme pour voir s'il correspondait bien aux attentes.

Chapitre 2

Protocole de groupe et de diffusion

2.1 Présentation générale

Il s'agit là d'un programme de protocole de groupe et de diffusion à ordre total tolérant aux fautes. Ces notions seront détaillées un peu plus loin, mais on peut dire en résumé qu'il permet de s'inscrire à un groupe, ou d'en constituer un, et de communiquer avec ses membres de manière à maintenir un état commun (la composition du groupe et/ou la valeur d'une variable par exemple), et qu'il est tolérant aux fautes, c'est à dire aux pannes ou crash de processus.

Il est utile aux applications coopératives distribuées qui s'affranchissent de plus en plus d'un contrôle centralisé en faveur d'un contrôle distribué permettant de pallier plus facilement aux pannes du contrôleur ou du réseau de communication, d'éviter les problèmes de congestion, etc.

Qui plus est ce programme présente l'avantage d'être compact, il est donc très portable. Et son découpage modulaire lui permet également de s'adapter facilement à différentes interfaces réseau.

Pour comprendre ce logiciel et son intérêt, deux choses sont à définir : le membership (ou protocole de groupe) et l'atomic broadcast (autrement appelé diffusion à ordre total) qui sont les deux principes de base du programme.

2.1.1 Protocole de groupe

Un protocole de groupe permet entre autre de :
- s'inscrire à un groupe (ou le créer s'il n'existe pas)
- communiquer avec les autres membres (envoyer et recevoir des messages)
- connaître la composition du groupe

– et enfin quitter le groupe.

Parmi les propriétés recherchées nous avons :

– tous les processus d'un même groupe ont la même vision du groupe
– les messages de modification du groupe sont délivrés dans le même ordre à tous les membres du groupe
– les messages de modification du groupe se situent au même endroit dans le flux des messages pour tous les membres du groupe

Ces propriétés sont intimement liées car elles s'impliquent mutuellement.

2.1.2 Diffusion à ordre total

L'atomic broadcast a entre autre comme propriétés (les plus intéressantes dans notre cas) :

– si un processus reçoit un message m cela implique que tous les processus reçoivent m
– si deux processus reçoivent deux messages m1 et m2, alors ils les recevront dans le même ordre.

2.1.3 Résistance aux fautes

Ce logiciel ayant pour but d'être résistant aux fautes, cela signifie qu'on veut donc en plus intégrer des pannes (de sites et de réseau) ce qui en vertu de travaux dans ce domaine est impossible à réaliser sans restreindre les hypothèses. En effet on rencontre sinon des problèmes d'indécidabilité en cas de pannes, de pertes de messages et/ou de crash de machines dans un contexte où une borne au délai de transmission des messages ne peut être assurée, ce qui est le cas ici puisque le programme utilise un protocole UDP qui ne garantit pas la délivrance des messages.

2.2 Présentation de l'algorithme

Le fonctionnement du logiciel est basé sur la notion de phase. Au cours d'une phase :

– les participants s'échangent des messages et ils ne sortent de la phase que lorsque tous les messages de cette phase ont été reçus. De cela il découle une propriété qui veut que même si tous les participants ne sont pas dans la même phase au même moment, il y a au plus une phase de différence entre deux d'entre eux (c'est à dire qu'à un instant donné, quel qu'il soit, il n'existe que deux types de phase, i et i+1).
– des requêtes peuvent également être envoyées aux membres dont le message n'a pas encore été délivré, elles résultent de l'arrivée à échéance d'un timeout et elles permettent de pallier le problème de possible perte de messages dû à l'utilisation d'un protocole UDP. Il y a donc une

deuxième façon de sortir de la phase : l'arrivée a échéance du timeout global, ce qui traduit l'existence de processus fautifs.

– les participants prennent une décision sur l'état du groupe à la phase précédente à partir des messages reçus à la phase courante ceux ci étant en effet les bilans de la phase précédente faits par chaque membre. Si les hypothèses de correction du protocole sont satisfaites tous les membres prennent la même décision quant à l'état du groupe : ils ont donc la même vision du groupe.

L'hypothèse de correction de cet algorithme est qu'il n'y a pas deux phases consécutives avec des erreurs. En effet il a été montré que s'il y a eu n phases avec des membres fautifs, il ne peut y avoir consensus qu'à la $(n+1)^{ème}$ phase.

C'est cette hypothèse qui permet de pallier partiellement au problème d'indécidabilité.

Chapitre 3

Présentation du programme

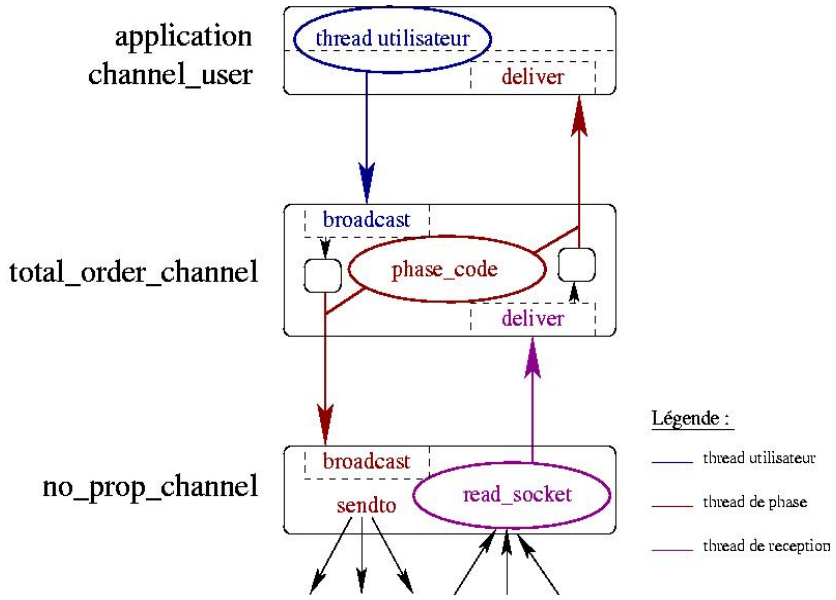

FIG. 3.1 – Schéma du programme

3.1 Explication du fonctionnement et du code du programme

3.1.1 Intégration au groupe

Pour intégrer un groupe, il y a deux procédures possibles, selon que le groupe existe ou non :

– si le groupe n'existe pas alors on est initiateur. On doit donc créer le groupe à partir d'un fichier dans lequel on trouve son identificateur,

son adresse IP et son numéro de port. Ensuite le fait d'être le seul membre du groupe ne pose pas de problème au bon déroulement du programme.
- si le groupe existe déjà, toute personne qui souhaite l'intégrer doit envoyer un message de type NEW à un membre dont il a récupéré les coordonnées dans un fichier. Ce dernier à la phase suivante envoie un message de type JOIN à l'ensemble du groupe. En réaction chaque membre augmente le timeout de sa phase afin de laisser au nouveau venu le temps de s'intégrer, intègre le nouvel arrivant en lui attribuant un identificateur (qui sera le même pour tous les membres du groupe car les messages sont délivrés dans le même ordre) et en dehors de la phase lui envoie un message de type INIT, lui transmettant ainsi l'identificateur qu'il lui a attribué et les données du groupe. Ensuite à la phase suivante, tout le monde a mis le groupe à jour et en a la même vision.

On peut diviser le programme en deux classes C++ et une application (le programme principal) (cf. fig. 3.1) qui ont chacun une tâche bien spécifique :
- `appli` qui comme son nom l'indique est l'application dans laquelle l'utilisateur peut joindre un groupe puis communiquer avec les autres membres de ce groupe (dans un chat par exemple).
- la classe `total_order_channel` qui est la plus grosse partie du code et du processus. Elle contient notamment les méthodes `deliver` et `phase_code` qui gèrent respectivement les messages sortants pour l'un, et les messages entrants et les réactions du système en présence d'un message pour l'autre.
- la classe `no_prop_channel` qui est quant à elle la couche la plus proche du réseau : elle gère l'accès au réseau c'est à dire l'initialisation, l'envoi et la réception de messages.

Par ailleurs ce programme se compose de trois threads (un thread étant un flux de calcul), un thread utilisateur qui est le thread parent, celui de départ. À partir de celui-là on crée un thread de réception qui est chargé de recevoir les messages et de les analyser grâce à la méthode `deliver`, et si besoin est de réagir (si il s'agit d'un NEW ou d'un INIT), et ensuite on crée un thread de phase qui comme son nom l'indique gère la phase, c'est à dire les réactions en fonction du comportement du groupe et les timeouts.

3.1.2 L'utilisateur

Il appelle le constructeur de la classe `total_order_channel` qui réagit de manière différente selon que l'on est initiateur ou suiveur. Il initialise les données de l'utilisateur (son adresse IP, son numéro de port et son groupe). Pour cela il appelle le constructeur de la classe `no_property_channel` ainsi que les différentes méthodes correspondantes. Dans `no_property_channel` il crée également le thread de réception. Enfin, de retour dans `total_order_`

`channel`, il crée le thread de phase.

3.1.3 Le thread de réception

Ce thread commence par la méthode `read_socket` qui ouvre une socket UDP de réception qui reste en permanence a l'écoute et permet d'appeler la méthode `deliver` dès qu'elle reçoit un message.

Détaillons maintenant la méthode `deliver` qui est le processus d'analyse des messages reçus par la socket d'écoute. Les messages reçus sont dans un premier temps triés selon la phase de l'expéditeur, puis selon le type du message :

- Si le message provient de la phase précédente, on l'ignore (tout comme si par accident il venait d'une phase ayant plus de 1 de décalage avec notre phase courante) sauf s'il s'agit d'un NACK auquel cas on renvoie le message qu'on a envoyé lors de la phase précédente.
- s'il provient de la phase courante alors on trie selon le type du message et on remplit en conséquence le buffer de phase de la phase courante (c'est à dire l'état de chaque membre du groupe, le message qu'il a envoyé, s'il en a envoyé un et le nombre de message de chaque type reçus).
- Si le message vient de la phase suivante, on stocke le message et on remplit donc le buffer de phase de la phase suivante.

Les seuls types de messages différenciés sont le NEW et l'INIT :

- si on reçoit un NEW, on fait appel à la méthode `start_join` qui rédige un CMD de type JOIN qui sera envoyé à l'ensemble du groupe.
- pour ce qui est de l'INIT, on n'en tient compte que si on a le statut PRERUN, c'est à dire qu'on a envoyé un NEW, mais qu'on n'a pas encore reçu le message d'initialisation, on n'est donc pas encore dans le groupe. Auquel cas en recevant l'INIT on appelle `new_member_init` qui fait l'initialisation grâce aux données contenues dans le message.

3.1.4 Le thread de phase

Ce thread se compose principalement d'une méthode, `phase_code` qui comme son nom l'indique, gère la phase c'est à dire les réactions aux messages, les timeouts et l'envoi des messages.

`phase_code` se déroule ainsi (cf. fig. 3.2) :

- envoi d'un message en attente suivant le type :
  - si message de type FAULT (message contenant l'ensemble des processus déclarés fautifs, c'est à dire ceux dont on n'a pas reçu de message à la phase précédente) envoi de ce message
  - sinon si message de type CMD (JOIN ou QUIT) (message indiquant une demande d'adhésion ou de résiliation d'un membre) envoi de ce message

  - sinon si message de type MESS (message de l'application) envoi de ce message
  - sinon si message de type ACK (message vide signifiant simplement l'acquittement de la phase précédente) envoi de ce message
- si on n'a pas reçu tous les messages de la phase courante et que le timeout arrive à échéance : appel de la méthode `timeout_handler` pour envoyer un NACK (message de non-réception) aux membres dont on n'a pas reçu de message
- si on a reçu tous les messages de la phase ou que le troisième timeout est écoulé on sort de la phase
- Si le timeout est écoulé (c'est à dire qu'on n'a pas reçu tous les messages attendus) : appel de la méthode `fault_handler` (elle supprime les membres pour lesquels on est toujours en attente, et rédige un message de type FAULT qui sera envoyé à la phase suivante)
- appel de la méthode `phase_actions` (elle analyse le buffer de phase et réagit en fonction) :
  - parcourt de la liste des membres, si un membre a été déclaré fautif, il est supprimé (du buffer de phase suivant et du groupe)
  - parcourt de la liste un seconde fois, si un CMD a été reçu :
    - si c'est un JOIN) : parcourt de tout le groupe jusqu'au premier identificateur indéfini rencontré, qui est alors déclaré alors comme étant un nouveau membre et est renseigné avec les données du JOIN
    - si c'est un QUIT : suppression du membre concerné
  - parcourt de la liste une dernière fois, s'il y a des nouveaux membres : appel de la méthode `new_member_init` qui rédige un message de type INIT qui est envoyé au nouveau membre et ensuite modification de son statut de manière à l'identifier comme étant intégré.
- retour dans `phase_code`, appel de la méthode `phase_update` qui met à jour les buffers de phase pour passer à la phase suivante.

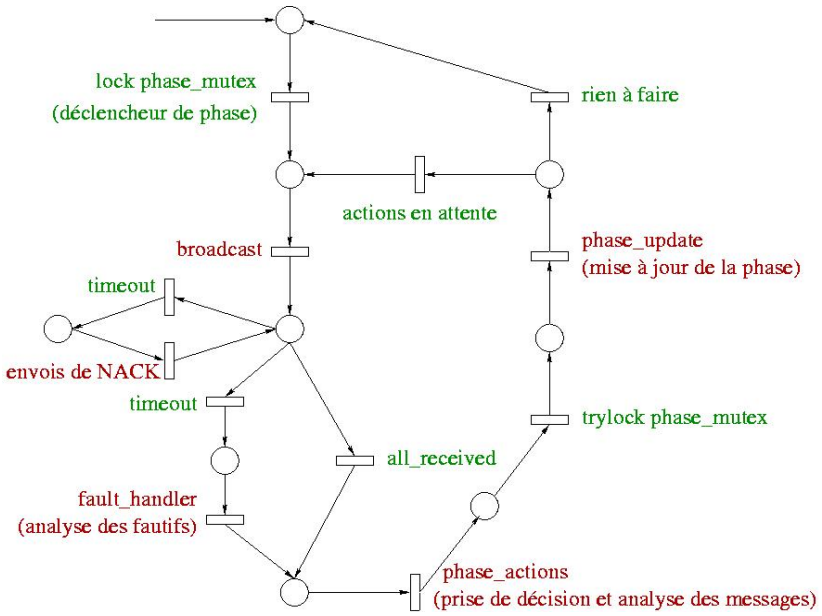

FIG. 3.2 – Déroulement d'une phase

3.2 Les threads

Les threads comme dit précédemment sont des flux de calcul, ils permettent de mener plusieurs activités différentes de manière simultanée. Ils s'organisent de manière filiale, c'est à dire qu'on a un thread parent qui engendre les autres, il faut noter que si le thread parent meurt, les threads descendants feront de même. Les threads partageant certaines zones mémoire, on doit utiliser des sémaphores (ou mutex) pour gérer cela. Dans notre cas l'envoi et la réception de messages sont respectivement communs aux threads phase et utilisateur et aux threads phase et réception. Les mutex fonctionnent ainsi : ils permettent de bloquer une zone mémoire afin d'empêcher deux routines de lire ou d'écrire dessus au même moment. Le mutex peut prendre deux valeurs, 0 ou 1 s'il est à 0 cela signifie qu'il est libre et donc inversement, s'il est à 1 alors il est occupé. On manipule les mutex ainsi : on commence par demander le mutex, s'il est à 1, alors on attend qu'il passe à 0, c'est à dire qu'il soit libéré, dés qu'il est à 0 on le met à 1 pour le bloquer, on fait ce qu'on a à faire puis on le libère, c'est à dire qu'on remet sa valeur à 0.

3.3 Adresses IP et sockets

Ici la manipulation des adresses IP et des sockets se fait via un protocole UDP, qui ne garantit ni l'intégrité, ni l'ordre des messages. Les sockets quant à elles sont les interfaces avec le réseau, elles permettent l'envoi et la réception de messages grâce à une adresse IP et un numéro de port. La manipulation des sockets se limite à l'ouverture, l'initialisation et la fermeture.

3.4 Participation personnelle

Ma participation à l'élaboration de ce programme a consisté à modifier la partie interface avec le réseau de manière à ce qu'elle n'utilise plus un protocole UDP mais multicast, afin de simplifier et de clarifier l'intégrations à un groupe et les échanges. En raison de la division en classe, cela s'est essentiellement résumé à remplacer la classe `no_prop_channel` par une nouvelle classe baptisée `multicast_channel` qui n'utilise plus un protocole UDP, mais un protocole multicast, les modifications à faire dans `total_order_channel` furent mineures cela résultant de la modularité du programme. Il s'est donc agi de remplacer une interface réseau par une autre.

Chapitre 4

Protocole réseau : Multicast

4.1 Présentation du multicast et intérêt dans notre cas

4.1.1 Présentation

Dans le cas du protocole UPD unicast, les messages sont envoyés directement au destinataire, on a donc besoin d'une adresse IP et d'un numéro de port pour chaque destinataire et si on doit envoyer un message identique à plusieurs processus, on doit l'envoyer à chacun séparément. Le protocole multicast quant à lui est un protocole UDP particulier, car il simplifie le processus : au lieu d'envoyer plusieurs versions du message, on envoie un seul message à l'adresse du groupe multicast et c'est le protocole réseau qui se charge de transmettre le message à l'ensemble des membres. Pour recevoir les messages, on doit d'abord s'inscrire au groupe multicast, ce qui n'est pas nécessaire pour envoyer des messages aux membres inscrits. Cela simplifie donc les procédures d'envoi et de réception. L'adresse multicast est une adresse IP classe D, il s'agit d'une adresse immatérielle dite adresse de groupe, elle est comprise dans la gamme allant de 224.0.0.0 à 239.255.255.255, mais pour des causes de réservations, il faut restreindre cette gamme de 224.0.0.255 à 239.0.0.0 en effet les autres adresses sont déjà utilisées. Ensuite on se rejoint ou on quitte le groupe grâce à une option de la socket de réception que l'on modifie grâce à la commande setsockopt(). Cette commande permet également de modifier les options d'une socket. Notamment le TTL (time-to-live) qui permet de restreindre ou d'étendre le champs d'émission des messages. Ensuite l'échange de message se fait comme pour un protocole UDP unicast grâce à sendto() et recvfrom() la seule chose qui change réside donc dans l'initialisation des sockets et dans les options.

4.1.2 Intérêt

Le multicast permet donc d'envoyer un même message à plusieurs utilisateurs tout en n'ayant à manipuler et donc à connaître qu'une adresse IP et un numéro de port, ceux du groupe multicast. On n'a donc plus besoin de connaître et donc de stocker les adresses IP et les numéros de port individuels des membres du groupe. J'ai choisi de les conserver afin de ne pas trop modifier `total order_channel` et ainsi conserver la modularité du programme, mais ils ne sont plus utiles. Le seul moment où on manipule ce genre de données personnelles est lors de l'envoi d'un unicast ce qui n'a lieu que lors de l'initialisation d'un nouveau membre et dans ce cas, on transmet ces données à partir du JOIN reçu.

Le multicast simplifie aussi l'intégration au groupe, en effet on n'a plus à connaître les coordonnées d'un membre, il suffit de connaître celle du groupe, on peut envisager d'attribuer au groupe un identifiant permettant ainsi de récupérer ses coordonnées dans un fichier.

Le broadcast est également simplifié puisqu'on n'a plus à envoyer les messages à chaque destinataire un par un, le processus est donc plus rapide.

4.2 Code test

Avant de modifier le code, pour comprendre le multicast j'ai rédigé un programme test très basique qui a juste pour but de tester le multicast et de me familiariser avec son codage. Afin de pouvoir ensuite l'appliquer à notre cas.

4.2.1 Code initial

Le programme est très simple. Il est constitué d'un main, et prend en entrée le numéro de port, le nombre de messages que l'on souhaite envoyer et un identificateur qui aidera à mieux analyser les affichages obtenus lors des tests. L'adresse IP et le numéro de port du multicast sont eux en dur dans le programme pour plus de facilité.

Le programme commence par ouvrir deux sockets permettant de communiquer avec le réseau, une socket d'émission pour envoyer des messages et une socket d'écoute pour en recevoir.

Ensuite il les initialise c'est à dire, il renseigne la famille des sockets, leurs adresses et leur numéro de port et calcule leur taille. Les deux sockets font partie de la famille AF_INET, leur port est celui du groupe multicast la différence se situe au niveau de l'adresse IP, celle de la socket d'émission est celle du groupe, mais celle de la socket de réception est une adresse par défaut (INADDR_ANY). Pour cette dernière il doit en plus faire un bind() avec son descripteur. Ensuite afin de pouvoir recevoir les messages du groupe, il s'inscrit au groupe multicast en modifiant l'option IP_ADD_MEMBERSHIP de

la socket de réception. Il modifie aussi le TTL de la socket d'émission afin d'étendre le champ de communication.

Après cela il fait deux boucles une pour émettre et l'autre pour recevoir les messages. La première se fait sur le nombre de messages émis. La seconde quant à elle est une boucle infini.

4.2.2 Tests

Dans un premier temps, j'ai testé avec un seul membre, tout se passe bien, sauf qu'à cause de la boucle infinie, on est obligé de crasher le processus pour pouvoir le terminer. Pour remédier à cela il suffirait de mettre un timout à la fin. Mais hormis cela, tous les messages sont bien émis et reçus.

Si on ajoute maintenant un nouveau membre alors que le premier a envoyé et reçu tous ses messages, tout va bien, il reçoit tous les messages du nouvel arrivant. En revanche, si on lance plusieurs processus en même temps, un très grand nombre de messages sont perdus. D'ailleurs plus on augmente le nombre de messages émis, plus la proportion de messages perdus augmente. On explique cela par la rapidité du processus et le chevauchement des tâches et des informations. On observe par exemple que si un processus reçoit des messages d'un nouveau membre alors qu'il est en train de recevoir les siens, il abandonne la réception de ses propres messages au profit de ceux du nouvel arrivant. De même les processus ont tendance ne pas recevoir les derniers messages qu'ils ont envoyés alors qu'ils sont reçus par les autres processus.

Pour remédier à tout cela j'ai temporisé l'envoi des messages en insérant des `sleep()` avec un certain intervalle. J'ai également séparé l'émission de la réception en créant un thread de réception

4.2.3 Modifications et conséquences

La quasi-intégralité du code final est en annexe A

La première modification est la création du thread de réception. On laisse l'ouverture et l'initialisation de la socket de réception dans le main, mais on doit sortir la boucle de réception pour la mettre dans une fonction séparée qui prend obligatoirement un void * en entrée. Pour cette raison, on met en entrée un argument qui ne sera pas utilisé par la suite. On met également en dehors des fonctions toutes les variables concernant les sockets afin de pouvoir les manipuler aussi bien dans le main que dans la fonction. On insère donc la création du thread ainsi que le test qui l'accompagne à la ligne 109 :

```
  int tlle;
  int ret = pthread_create(&read_thread,0,recept,
(void *) &tlle);

  if (ret)
    {
```

```
    perror(" no-prop() ");
    printf(" erreur th create  %d\n",ret);
    if (ret == ENOMEM) printf(" ENOMEM \n" );
    if (ret== EINVAL) printf(" EINVAL \n" );
    if (ret == EPERM) printf(" EPERM \n" );
    exit(0);
  }
```

Ensuite on place la temporisation dans la boucle d'émission (lignes 143-144) ici on fait une temporisation tous les 10 messages grâce à un modulo que l'on peut modifier selon le nombre de messages que l'on souhaite envoyer.

On fait également une troisième modification : dans les options de la socket d'émission, on retire l'auto-bouclage ce qui permet de ne pas recevoir ses propres messages. En effet une copie automatique des messages émis étant faite dans le programme, on n'a pas besoin de les recevoir : ce serait redondant. Pour pouvoir utiliser une adresse IP pour plusieurs membres d'un même groupe, on doit faire :

```
int reuse=1;
if (setsockopt(sdr,SOL_SOCKET,SO_REUSEADDR,(int
*)&reuse,sizeof(reuse) == -1)
    perror("setsockopt : SO_REUSEADDR");
```

et dans ce cas, il s'avère qu'il faut laisser l'auto-bouclage, sinon les membres utilisant la même adresse IP ne pourront pas communiquer entre eux.

4.3 Modification du code

Les modifications concernent le protocole réseau, elles s'opèrent surtout au niveau de `no_prop_channel`, qui devient d'ailleurs par-là même `multicast_channel`, mais également au niveau de l'intégration d'un nouveau membre, de la classe `group_admin` et de la communication (broadcast).

4.3.1 group_admin

On doit modifier la classe `group_admin`. En effet la nature du groupe a changé, il s'agit maintenant d'un groupe multicast, il prend donc en compte deux attributs de plus, que sont l'adresse IP et le numéro de port. Le premier étant un pointeur sur une chaîne de caractère et le second un entier. À ces deux attributs correspondent donc des méthodes qui sont : un nouveau constructeur prenant en arguments l'adresse IP et le numéro de port, deux méthodes permettant de récupérer ces attributs ainsi que deux autres permettant de les initialiser.

Les nouveaux attributs et méthodes cette classe sont en annexe B

4.3.2 Intégration au groupe

Pour modifier l'intégration au groupe, on est obligé de modifier à la fois `multicas_channel` et `total_order_channel`, on modifie les méthodes déjà existantes, mais on en crée aussi de nouvelles. Pour montrer ces modifications, voyons pour chaque cas l'ancien processus et le nouveau.

Pour avoir une vision d'ensemble de l'initialisation suivant que l'on est initiateur ou suiveur, une partie du code du constructeur de `total_order_channel` est en annexe C.

Initiateur

Tout se passe comme avant jusqu'à l'appel du constructeur de `multicast_channel` approprié.

Avant :

- on appelait le constructeur de `no_prop_channel` qui prenait en entrée un pointeur sur un objet de type `total_order_channel` (construit précédemment grâce à la méthode `init_data`, qui initialise toutes les données), l'identificateur, un nom de fichier (donné afin de distinguer les deux constructeurs), l'adresse IP et enfin le groupe de l'individu. Cette méthode :
  - initialise les données de `no_prop_channel`
  - crée le thread de réception qui appelle `read_socket` qui ouvre la socket UDP de réception et l'initialise, puis qui reçoit les messages et les transmet à `deliver`

Ensuite on crée le thread de phase, mais tout cela reste identique.

Maintenant : le schéma d'action est le même excepté que :

- le constructeur de `multicast_channel` n'est plus le même, il prend toujours les mêmes arguments excepté que l'adresse IP transmise n'est plus celle de l'individu, mais celle du groupe, et il fait toujours les mêmes choses, mais en plus il ouvre une socket d'émission multicast qu'il initialise et qui restera ouverte toute la durée du programme, cela évite d'avoir à en créer une à chaque fois qu'on envoi un message puisque de toute manière on utilise toujours la même.
- de même la socket créée dans le thread de réception n'est plus une socket UDP, mais une socket multicast.

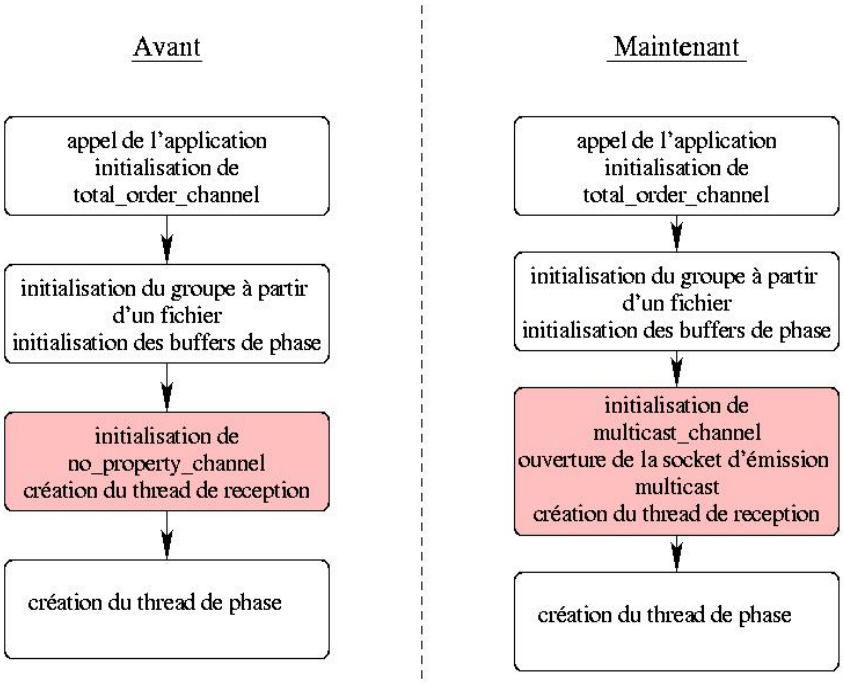

FIG. 4.1 – Modification du comportement d'un initiateur

Le constructeur de `multicast_channel` et `read_socket` sont en annexe D.

Suiveur

Avant :

Le nouveau membre récupérait dans un fichier l'adresse IP et le numéro de port d'un membre du groupe à qui il envoyait un message de type NEW et il attendait la réponse (l'INIT) sur la socket UDP de réception. Cela se traduisait ainsi :

- appel au constructeur de `no_prop_channel` approprié qui prend en entrée un pointeur sur un objet de type `total_order_channel`, l'adresse IP de l'utilisateur et son groupe. Il initialise les données de manière temporaire puis crée le thread de réception qui ouvrira la socket UDP de réception .
- appel à la méthode `start_net_init` qui prend en entrée le nom d'un fichier qui contient l'adresse IP et le numéro de port d'un membre du groupe. Il crée un message type NEW qu'il transmet avec le nom du fichier à la méthode `init_membership` de `no_prop_channel` qui ouvre et initialise une socket UDP d'émission, envoie le message à l'adresse indiquée dans le fichier puis referme la socket. Ensuite l'utilisateur a le statut PRERUN jusqu'à ce qu'il reçoive un INIT.

Maintenant :

On utilise maintenant des sockets multicast pour la plupart des communications, on n'a donc plus besoin des coordonnées d'un membre du groupe pour pouvoir le joindre, on envoie un NEW via la socket multicast (ce qui se passe du côté de l'interlocuteur sera détaillé plus loin), et on attend la réponse sur une socket particulière qui elle est une socket UDP, en effet la réponse ne sera pas envoyée via le multicast. Cela se passe ainsi :

- appel du constructeur de `multicast_channel` approprié qui prend toujours les mêmes choses en entrée (dans ce cas l'adresse IP reste celle de l'utilisateur) et elle fait toujours les mêmes choses excepté que la socket de réception créée dans le thread du même nom est une socket multicast et non plus UDP.
- appel à `ouv_sock_INIT` qui est une méthode de `multicast_channel` qui ouvre une socket de réception UDP temporaire ayant pour seul but de recevoir le message d'initialisation.
- appel de `start_net_init` qui rédige un message de type NEW (contenant ses données personnelles, adresse IP et numéro de port permettant à son interlocuteur de lui renvoyer l'INIT via une socket UDP) et le transmet ainsi que les coordonnées du groupe multicast à la méthode `init_membership` de `multicast_channel`. Cette dernière ouvre une socket d'émission multicast qui restera ouverte puisque ce sera celle utilisée par la suite, puis elle envoie le NEW au groupe multicast. Ensuite une fois revenu dans le constructeur on initialise le statut à PRERUN, ce qui évite de tenir compte des messages reçus sur la socket de réception multicast.
- appel de `recept_INIT` qui est une méthode de `multicast_channel` qui attend (grâce à une boucle infinie) la réception d'un INIT et qui transmet les messages reçus à la méthode `receive_INIT` de `total_order_channel` avant de fermer la socket de réception temporaire (dés qu'un message de type INIT a été reçu). `receive_INIT` analyse le message, vérifie qu'il s'agit bien d'un INIT et auquel cas elle les transmet à la méthode `init_from_net` et elle initialise le statut à RUN.Si le message et bien un INIT, elle renvoie 0 ce qui permet de sortir de la boucle de `recept_INIT` et de fermer la socket, sinon elle renvoie 1. `init_from_net` initialise les données de l'utilisateur (grâce à l'identificateur transmis par le groupe), les données du groupe, et les buffers de phase.

Ensuite l'initialisation est terminée, l'utilisateur fait partie du groupe et il n'est plus distinguable des autres.

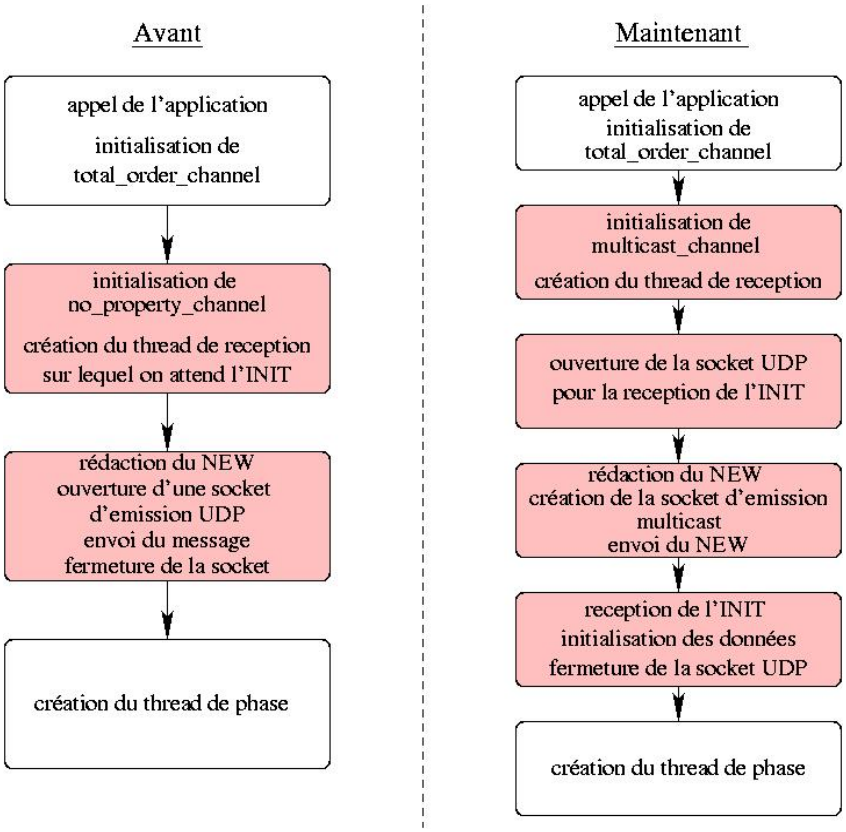

FIG. 4.2 – Modification du comportement d'un suiveur

Des parties de `init_membership`, `ouv_sock_INIT`, `init_membership`, `init_membership`, `recept_INIT`, et de `receive_INIT` sont en annexe E.

Initialiseur

Il s'agit de la réaction d'un membre du groupe face à l'arrivée d'un nouveau membre.

Avant :

Le nouveau membre envoyait un NEW à un membre du groupe qui envoyait à son tour un JOIN à l'ensemble du groupe afin de faire savoir l'arrivée d'un nouveau membre afin que chacun mette son groupe à jour, et attribue le même identificateur et envoie un INIT au nouveau venu.

Maintenant :

L'ensemble du groupe reçoit le NEW via le multicast, mais afin que tout le monde lui attribue le même identificateur, un membre doit se désigner (grâce à un test) pour envoyer un JOIN au groupe et ensuite la procédure reste la même qu'avant. Cela se passe ainsi :

- le NEW est transmis à `deliver` par la socket de réception, une fois identifié comme étant un NEW, `deliver` appelle la méthode `start_join`
- `start_join` fait le test qui dans notre cas consiste à voir qui est le premier membre de la liste et à voir si on est ce premier membre. Auquel cas on rédige un JOIN
- s'il n'y pas eu d'erreur à la phase précédente, le CMD étant le second dans l'ordre de priorité des message à envoyer dans phase code, on broadcast le JOIN
- à la phase suivante, tous les membres reçoivent le JOIN, dans le même ordre, ils intègrent le nouveau groupe en lui attribuant un identificateur et rédigent un INIT qu'ils envoient via unicast hors phase, il n'y a pas de modification dans cette dernière partie du processus c'est la raison pour laquelle elle n'est pas détaillée plus que cela ici.

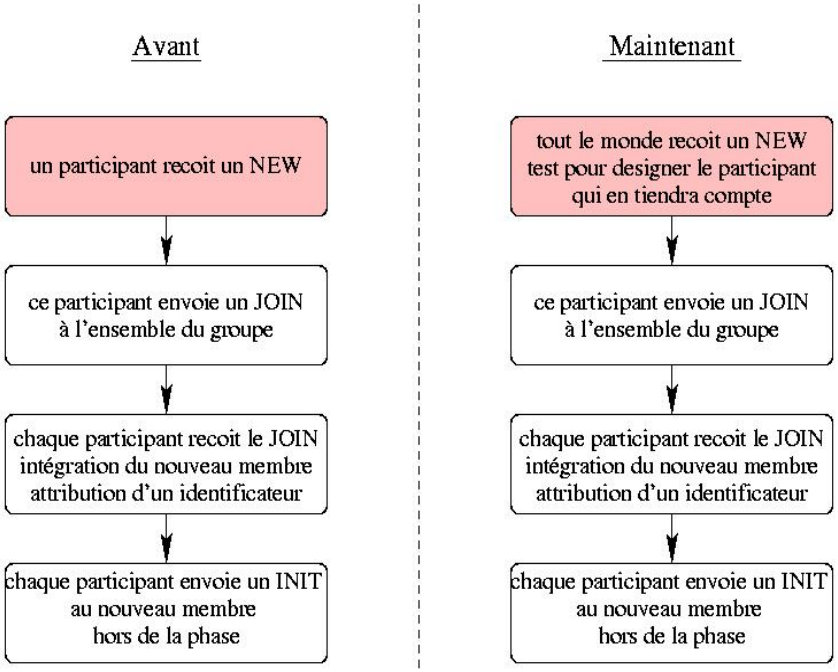
Avant
un participant recoit un NEW
ce participant envoie un JOIN à l'ensemble du groupe
chaque participant recoit le JOIN intégration du nouveau membre attribution d'un identificateur
chaque participant envoie un INIT au nouveau membre hors de la phase
Maintenant
tout le monde recoit un NEW test pour designer le participant qui en tiendra compte
ce participant envoie un JOIN à l'ensemble du groupe
chaque participant recoit le JOIN intégration du nouveau membre attribution d'un identificateur
chaque participant envoie un INIT au nouveau membre hors de la phase

FIG. 4.3 – Modification de l'initialisation d'un nouveau membre

`start_join` est en annexe F.

4.3.3 Envoi de messages

Avant :
Il y avait avant trois types d'envoi de message :
- le broadcast qui permettait d'envoyer à l'ensemble du groupe en faisant une boucle sur le nombre de membre dans laquelle on envoyait le

message à chaque membre séparément.
- le multicast qui permettait d'envoyer le message à une liste de membre (utilisé pour l'envoi de NACK) mais qui fonctionnait sur le même principe que le broadcast.
- enfin l'unicast qui permettait d'envoyer à une seule personne, il est notamment utilisé pour l'entrée d'un nouveau membre.

Ces types d'envois ont en commun de créer une socket temporaire à chaque fois qu'ils sont appelés.

Maintenant :

Le multicast permet d'envoyer les même message à plusieurs destinataires en l'envoyant à une seule adresse, l'adresse multicast. Pour cette raison on modifie les procédures d'envoi, on n'en conserve plus que deux, le broadcast et le multicast.

- le broadcast utilise la socket multicast déjà créée, cette méthode se contente donc d'envoyer le message à l'adresse multicast. Il est utilisé pour quasiment tous les envois sauf pour l'initialisation d'un nouveau membre.
- l'unicast se fait grâce à une socket UDP, on doit donc créer une socket UDP temporaire pour l'envoi du message que l'on ferme juste après.

`broadcast` est en annexe G.

4.3.4 Divers

Les autres modifications sont des modifications mineures qui résultent de celles décrites précédemment. Notamment au niveau des arguments de certaines fonctions. On a également modifié légèrement le `deliver`, on en a retiré l'analyse des INIT, en effet, il n'est pas reçu sur la même socket, on a préféré le distinguer et l'intégrer au processus d'intégration.

4.4 Fonctionnement et tests

4.4.1 Fonctionnement

Le déroulement entier du code est en annexe H.

En tant qu'initiateur

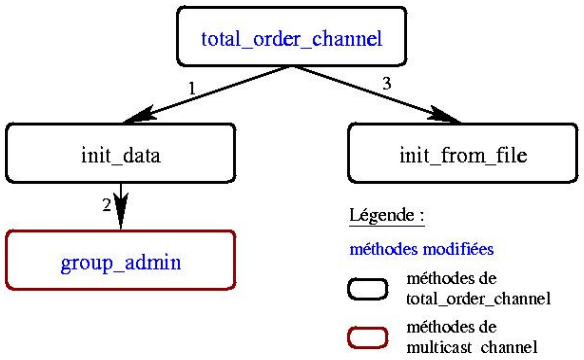

FIG. 4.4 – Déroulement de l'intégration pour un initiateur

En tant que suiveur

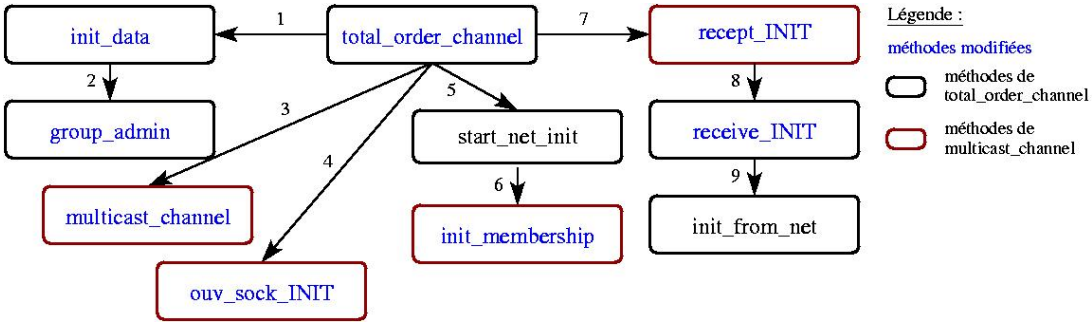

FIG. 4.5 – Comportement de l'intégration pour un suiveur

Mode d'emploi

Pour manipuler le programme on doit rédiger une application `appli` qui nous permettra de le tester. Ce programme prend en entrée l'option (0 si on est initiateur, 1 si on est suiveur), l'adresse IP individuelle si on est suiveur ou l'identifiant si on est initiateur, le nom d'un fichier ou si on est utilisateur

on récupère la liste des membres, le nombre n de messages que l'on souhaite envoyer, et enfin l'adresse IP et le numéro de port multicast du groupe. On lance le programme en appelant le constructeur de `total_order_channel` puis on fait une boucle qui permet de rédiger n messages de l'application.

Pour appeler cette application on fera donc (selon si on est respectivement initiateur ou suiveur) :

```
appli  0       1       voisin1.dat  100  238.1.6.6  55501
appli  1  140.93.0.13  voisin1.dat  100  238.1.6.6  55501
```

4.4.2 Tests

Hormis les tests de debuggage permettant de vérifier que le programme fait bien ce qu'on attend de lui c'est à dire émettre, recevoir des messages, détecter et déclarer un processus fautif, en accord avec les autres membres du groupe, il faut également tester la résistance du groupe. Pour tout cela j'ai réalisé différents tests me permettant de modifier et corriger le code de manière à ce qu'il réponde mieux aux attentes.

J'ai commencé par lancer un seul processus afin de voir si il ouvrait correctement les sockets, si son initialisation se passait correctement et s'il communiquait bien, même si ce n'est qu'avec lui même.

Puis j'ai lancé deux processus ensemble pour tester l'initialisation en tant que suiveur et pour voir si la communication se passait bien. Ensuite j'ai augmenté progressivement le nombre de processus pour voir si les crash étaient bien détectés, et si le consensus se passait bien.

Puis enfin j'ai lancé un grand nombre de processus en alternant les crash et les demande d'inscriptions pour tester les limites du programme, voir son comportement dans des conditions limites.

Chapitre 5

Conclusion

Ce stage m'a beaucoup apporté tant au niveau personnel qu'au niveau de mes connaissances. Au niveau des connaissances j'ai pu atteindre mon but qui était d'approfondir mes connaissances en informatique, je suis même allée au-delà de mes attentes car j'ai découvert des domaines qui m'étaient totalement inconnus, le protocole réseau et la programmation multi-threadée. Cela m'a également permis de mieux voir les applications de l'informatique et les problèmes que l'on peut rencontrer, au-delà de ce qu'on a vu en cours. Même si je n'ai pas particulièrement approfondi un domaine, cela m'a permis d'en aborder plusieurs, ce qui me semble plus intéressant.

Au niveau personnel ce stage m'a prouvé que je pouvais m'adapter à un domaine que je ne maîtrisais pas, cela m'a mise en confiance. Et le fait de réaliser une tâche qui ait une utilité et une application est quelque chose de particulièrement valorisant et motivant.

Ce stage étant prévu comme stage de DEA, le développement de ce logiciel est loin d'être terminé et d'autres perfectionnements sont donc envisageables.

Annexe A

Programme test multicast

```
020   void * recept(void *);   /* thread routine */
021   thread_t read_thread;
022   int sdr,sdw;
023   sockaddr_in sock_r,sock_w;
024   int len_r,len_w;
025
026       /****************************************/
027       /****************************************/
028
029   // l'adresse IP, le numéro de port de la machine et
l'identificateur sont donnes en arguments
030   int main(int argc,char* argv[])
031   {
032     int noport,nbmess,id;

        // récupération des données dans argv

044     /*********************************************/
045
046     // ouverture de la socket de réception
047
048     sdw=socket(PF_INET,SOCK_DGRAM,0);
049     if(sdw<0)
050       {
051         perror("socket");
052         exit(1);
053       }

        //  de même ouverture de la socket d'écoute
```

```
       /*********************************************/
065
066     // initialisation de la socket d'émission
067
068     memset(&sock_w,0,sizeof(sock_w));
069     sock_w.sin_family=AF_INET;
070     sock_w.sin_port=htons(PORT);
071     sock_w.sin_addr.s_addr=inet_addr(GROUP);
072
073     len_w=sizeof(sock_w);
074
075     // initialisation de la socket d'écoute
076
077     memset(&sock_r,0,sizeof(sock_r));
078     sock_r.sin_family=AF_INET;
079     sock_r.sin_port=htons(PORT);
080     sock_r.sin_addr.s_addr=htonl(INADDR_ANY);
081
082     len_r=sizeof(sock_r);
083
084     if
(bind(sdr,(struct sockaddr *) &sock_r,sizeof(sock_r))<0)
085       {
086         perror("bind");
087         exit(1);
088       }
089
090     /*********************************************/
091
092     // intégration au groupe;
093
094     struct ip_mreq imr;
095
096     imr.imr_multiaddr.s_addr=inet_addr(GROUP);
097     imr.imr_interface.s_addr=htonl(INADDR_ANY);
098
099     if(setsockopt(sdr,IPPROTO_IP,
IP_ADD_MEMBERSHIP,(void *) &imr,sizeof(struct ip_mreq))<0)
100       {
101         perror("setsockopt - IP_ADD_MEMBERSHIP");
102         exit(1);
103       }
104
105     /*********************************************/
```

```
106
107     // création du thread de réception
108
109     int tlle;
110     int ret = pthread_create
(&read_thread,0,recept,(void *) &tlle);
111
112     if (ret)
113       {
114         perror(" no-prop() ");
115         printf(" erreur th create  %d\n",ret);
116         if (ret == ENOMEM) printf(" ENOMEM \n" );
117         if (ret== EINVAL) printf(" EINVAL \n" );
118         if (ret == EPERM) printf(" EPERM \n" );
119         exit(0);
120       }
121
122     /***********************************************/
123
124     // suppression de l'auto bouclage (loopback)
125     unsigned char loop=0;
126
127     if(r=setsockopt(sdw,IPPROTO_IP,
IP_MULTICAST_LOOP,&loop,sizeof(loop)))
129       perror("setsockopt : IP_MULTICAST_LOOP");
130
131     // envois des nbmess messages;
132     char buffer[300];
133     char * message=buffer;
134     unsigned char ttl=18;
135
136     if(setsockopt(sdw,IPPROTO_IP,
IP_MULTICAST_TTL,&ttl,sizeof(ttl))==-1)
137       perror("setsockopt : IP_MULTICAST_TTL");
138
139     for (int i=0 ; i<nbmess ; i++)
140       {
141         sprintf(message,"*** message no %d de %d***",i,id);
142         int j=strlen(message);
143         if (!(i%10))
144           sleep(1);
145         if
(sendto(sdw,message,j,0,(struct sockaddr *) &sock_w,len_w)<0)
146           {
```

```
147          perror("sendto");
148            exit(1);
149       }
150         cout << "envoi message " << i << " OK" << endl;
151       }
152
153    /*****************************************************/
154
155     sleep(20);
156
157     // quitter le groupe
158     if (setsockopt(sdr,IPPROTO_IP,
IP_DROP_MEMBERSHIP,&imr,sizeof(imr))==-1)
159       perror("setsockopt : IP_DROP_MEMBERSHIP");
160
161     close(sdr);
162     close(sdw);
163
164     return(0);
165   }
166
167    /*****************************************************/
168    /*****************************************************/
169
170   void * recept(void* tlle)
171   {
172     //réception  des messages du groupe
173     cout << "réception " << endl;
174     char buffer[300];
175     char * mess_r=buffer;
176     int t=0;
177     while(1)
178       {
179         int cnt;
180         cnt=recvfrom(sdr,mess_r,sizeof(buffer),0,
(struct sockaddr *) &sock_r,&len_r);
181         if(cnt<0)
182           {
183          perror("recvfrom");
184         exit(1);
185       }
186         else if(cnt==0) //fin de transmission
187        break;
188         t=t+1;
```

```
189         cout <<"message "<< t << ": " << mess_r << endl;
190       }
191   }
```

Annexe B

Classe group_admin

B.1 Le .h

```
class group_admin
{
public :
  //arguments
  int port;
  char * noIP;

  //méthodes
// constructeur avec numéro port et adresse IP
  group_admin(int ,char *);

  void set_gp_port(int);
  void set_gp_IP(char *);
//retrouver le numéro de port à partir d'un identifiant
  int get_gp_port();
  char * get_gp_IP(); //retrouver le no IP du gpe

#define MEMBSZ sizeof(struct member)
};
```

B.2 Le .cpp

```
group_admin :: group_admin(int no_port,char * addr_IP)
{
  groupsize=0;

  port=no_port;
  noIP=addr_IP;
```

```
  for (int i=0 ; i<MAX_MEMBER ; i++)
    members_list[i].ident=UNDEF;
}

/*******************************************************/

int group_admin::get_gp_port()
{return(port);}

void group_admin::set_gp_port(int pt)
{port=pt;}

/*******************************************************/

char * group_admin :: get_gp_IP()
{return(noIP);}

void group_admin::set_gp_IP(char * IP)
{noIP=IP;}
```

Annexe C

total_order_channel

```
total_order_channel :: total_order_channel
(int flag,char *file, char *ipnum,char *ipgp, char *port)
{
  if (flag == 1) // following member
    {
      strcpy(ipadd,ipnum);
      status = START ;
// initialisation
      net = new multicast_channel(this, ipnum, my_group);
      int ouv=net->ouv_sock_INIT();
// création de la socket d'émission et envoi du NEW
      start_net_init(gpIP,port_gp);
      status=PRERUN;
      ouv=net->recept_INIT(gpIP,port_gp);
    }
  else if (flag == 0) // first member
    {
      my_ident=atoi(ipnum);
      status = START;
      //remplissage de la liste de membres
      if (init_from_file(file))
{
  cout << "error opening " << file << endl;
  exit(0);
}
      status=RUN;
      net = new multicast_channel
(this, my_ident,file, my_group);
    }
}
```

Annexe D

Initiateur

D.1 Constructeur multicast_channel approprié

```
multicast_channel::multicast_channel(total_order_channel * pt,
 int id , char * file, group_admin  *gpt)
{
  cout << "start multicast channel 0 ( from file)" << endl;

  /*************************************************/
  /* initialisation*/

  my_user =  pt;

  my_group = gpt;

  my_ident = id;

  my_port = my_group->port_number(my_ident);

  char *IPgp=my_group->get_gp_IP();
  int port=my_group->get_gp_port();


  /***création et initialisation de la socket d'émission ***/

  sdw=socket(PF_INET,SOCK_DGRAM,0);
  if(sdw<0)
    {
      perror("socket");
      exit(1);
    }
```

```
  memset(&sock_w,0,sizeof(sock_w));
  sock_w.sin_family=AF_INET;
  sock_w.sin_port=htons(port);
  sock_w.sin_addr.s_addr=inet_addr(IPgp);


  unsigned char loop=0;
  int r=setsockopt(sdw,IPPROTO_IP,
IP_MULTICAST_LOOP,&loop,sizeof(loop));
  if (r==-1)
    perror("setsockopt : IP_MULTICAST_LOOP");

  // initialisation du TTL
  unsigned char ttl=18;
  if(setsockopt(sdw,IPPROTO_IP,
IP_MULTICAST_TTL,&ttl,sizeof(ttl))==-1)
    perror("setsockopt : IP_MULTICAST_TTL");

  /*** initialise input socket and start reading thread ***/


  int ret = pthread_create
(&read_thread, 0,start_thread, (void *)this);

  if (ret)
    {
      perror(" no-prop() ");
      printf(" erreur th create  %d\n",ret);
      if (ret == ENOMEM) printf(" ENOMEM \n" );
      if (ret== EINVAL) printf(" EINVAL \n" );
      if (ret == EPERM) printf(" EPERM \n" );
      exit(0);
    }
}
```

D.2 read_socket

```
int multicast_channel::read_socket()
{

  int ret , size, type,  fromlen;
  struct sockaddr_in client;
```

```
  int port = my_group->get_gp_port();
  char * IPgp = my_group->get_gp_IP();

  char message[64000];

  char * pt;
  pt = message;

  if (my_port == -1) my_port = BASEPORT;

  if ( (sdr=socket(PF_INET,SOCK_DGRAM,0) )<0)
    {
      cout << " erreur socket() start read socket " << endl;
      return(-1);
    }

  memset(&sock_r,0,sizeof(sock_r));
  sock_r.sin_family = AF_INET;
  sock_r.sin_port = htons(port);
  sock_r.sin_addr.s_addr = htonl(INADDR_ANY);

  if ( bind(sdr,
(struct sockaddr *)&sock_r,sizeof(sock_r)) < 0 )
    {
      perror("bind");
      exit(1);
    }

  // integration au groupe
  imr.imr_multiaddr.s_addr = inet_addr((const char *)IPgp);
  imr.imr_interface.s_addr = htonl(INADDR_ANY);

  //cout << "imr initialise" << endl;

  if(setsockopt(sdr,IPPROTO_IP,IP_ADD_MEMBERSHIP,
(void *) &imr, sizeof(struct ip_mreq))<0)
    {
      perror("setsockopt - IP_ADD_MEMBERSHIP");
      exit(1);
    }


  fromlen = sizeof sock_r;
  //cout << "intégration au groupe multicast OK" << endl;
```

```
  for(;;)
    {
      ret =recvfrom
(sdr, pt, 64000, 0,(struct sockaddr *) &sock_r, &fromlen );
      //cout << "message recu " << endl;
      if( ret == -1 )
{
  cout << " erreur recvfrom() start read socket " << endl;
  return(-3);
}

      my_user->deliver(pt, ret);
    }

  return(0);
}
```

Annexe E

Suiveur

E.1 ouv_sock_INIT

```
int multicast_channel::ouv_sock_INIT()
{
 // création de la socket de réception UDP  temporaire

  cout << "réception  de l'INIT " << endl;

  if (my_port == -1)
    my_port = BASEPORT;


  if ( (sdr_tp=socket(AF_INET,SOCK_DGRAM,0) )<0)
    {
      //création d'une socket UDP
      cout << " erreur socket() start read socket " << endl;
      return(-1);
  }

  sock_r_tp.sin_family = AF_INET;
  sock_r_tp.sin_port = htons(my_port);
  sock_r_tp.sin_addr.s_addr = htonl(INADDR_ANY);

  int iter = 1;
  do
    {
      if ( bind(sdr_tp, (struct sockaddr *)&sock_r_tp,
sizeof(sock_r_tp)) == -1 )
{
  my_port ++;
```

```
  sock_r_tp.sin_port = htons(my_port);
  iter ++;
  continue;
}
      else break;
    } while ( iter < 10 )  ;
  if (iter == 10 )
    {
      cout << " erreur  start read socket, " << endl;
      exit (2);
    }

  return(0);
}
```

E.2 init_membership

```
int multicast_channel::init_membership
(char *gpIP,int gpPort,char *ch,int sz)
{
  FILE *f;
  int ret,test;
  int sock, host, port;
  char ip[15];
  struct sockaddr_in nom;
  struct hostent *hp;


  // ouverture de la socket d'émission
 // (voir code de multicast_channel de l'initiateur)

  // on envoie le message
  if ((test=sendto(sdw, ch, sz, 0, (struct sockaddr *) &sock_w,
 sizeof sock_w)!=sz))
    {
      cout << "erreur sendto, sz = " << sz
<< " res = " << test << endl;
      return(-3);
    }

  // on laisse la socket ouverte
  // de manière à ne pas avoir à l'ouvrir en permanence
  return 0;
```

```
}
```

E.3 recept_INIT

```
int multicast_channel::recept_INIT(char * IPgp,int port)
{
  // réception de l'INIT

  int  size, type,  fromlen,ret;
  struct sockaddr_in  client;
  char message[64000];

  char * pt;
  pt = message;

  fromlen = sizeof client;
  int res=1;
  do
    {
      ret=recvfrom
(sdr_tp, pt, 64000, 0, (struct sockaddr *) &client, &fromlen );
      if( ret == -1 )
{
  cout << " erreur recvfrom() start read socket " << endl;
  return(-3);
}
      cout << "INIT recu "<< endl;
      res=my_user->receive_INIT(IPgp,port,pt, ret);
    }while(res!=0);

  close(sdr_tp);

}
```

E.4 receive_INIT

```
int total_order_channel::receive_INIT
(char * IPgp,int port,char *data, int size)
{
  int ret;
  int sender,mess_phase,type,mess_size;
  int indent,faulty;
  header *head = (header *) data;
```

```
  sender = head->sender;
  type = head->type;
  mess_phase = head->size;
  mess_size=head->size;
  if (mess_size != size-HEADSZ)
    cout << "** Problem of message size, got :" << size-HEADSZ
<< "expected :" << mess_size << endl;

  char * datapt = data+HEADSZ;
  if (type == INIT)
    {
      init_from_net(IPgp,port, datapt, mess_size );
      status = RUN;
      return(0);
    }
  else
    return(1);
}
```

Annexe F

Initialiseur

F.1 start_join

```
int total_order_channel::start_join(char *data,int size)
{
  header *head = &cmd_buf.head.head;
  cmdheader *cmdpt = &cmd_buf.head.chead;

  // on teste savoir si on est concerné
  // i.e. si on est le premier de la liste
  int test;
  for (int i=0 ; i < MAX_MEMBER ; i++)
    {
      if (cur_get->status[i] != UNDEF)
{
  test=i;
  break;
}
    }
  if (test == my_ident)
    {
      cout << "je suis le premier de la liste" << endl;

      if (cmd_to_send ==0 && init_flag == 0)
{
  head->sender = my_ident;
  head->num_phase = -1;
  head->type = CMD;
  head->size = CMDHEADSZ;

  cmdpt->type = JOIN ;
```

```
  cmdpt->ident = 0;
  cmdpt->nodeid = *(int *)data;
  cmdpt->port = *(int *)(data+sizeof(int));
  cmd_message.size = head->size;
  cmd_to_send = 1;
  init_flag = 1;
}
      else
cout << "start_join aborted : *** " <<
my_ident << "***" << endl;
    }
  else
    cout << " Je NE suis PAS le premier de la liste " << endl;
}
```

Annexe G

Envois de messages

G.1 broadcast

```
int multicast_channel::broadcast(char * ch,int sz)
{
  // la socket d'émission est déjà ouverte
  if(sendto(sdw,ch,sz,0,(struct sockaddr *) &sock_w,
sizeof sock_w)<0)
    {
      perror("sendto");
      exit(1);
    }
}
```

Annexe H

Fonctionnement du programme

H.1 En tant qu'initiateur

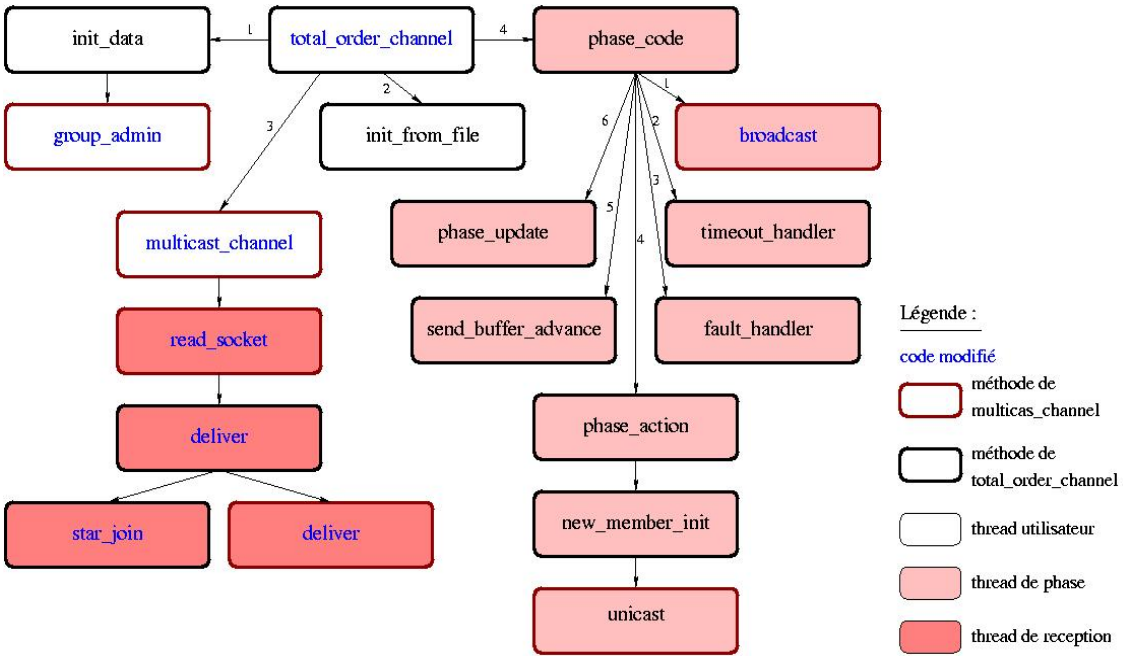

FIG. H.1 – Déroulement du programme pour un initiateur

H.2 En tant que suiveur

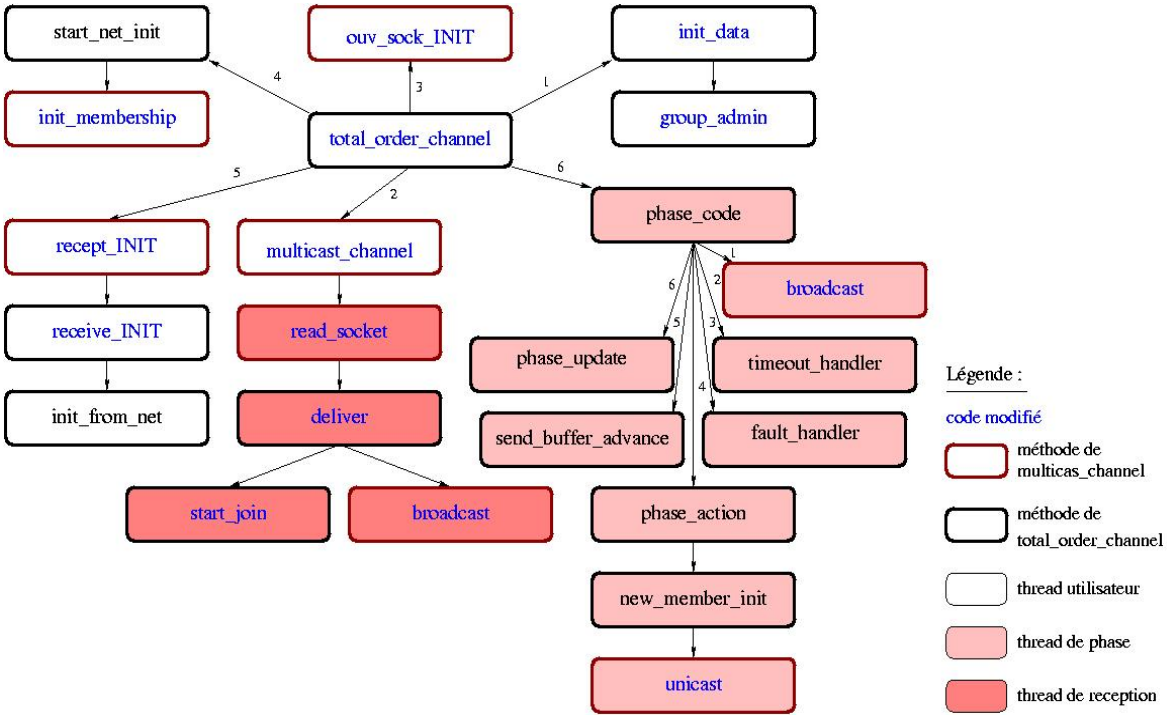

FIG. H.2 – Comportement du programme pour un suiveur